**Type conversions**

**Implicit conversion**

Implicit conversions are automatically performed when a value is copied to a compatible type. For example:

|  |  |  |
| --- | --- | --- |
| 1 2 3 | short a=2000;  int b;  b=a; |  |

Here, the value of a is promoted from short to int without the need of any explicit operator. This is known as a *standard conversion*. Standard conversions affect fundamental data types, and allow the conversions between numerical types (short to int, int to float, double to int...), to or from bool, and some pointer conversions.  
  
Converting to int from some smaller integer type, or to double from float is known as *promotion*, and is guaranteed to produce the exact same value in the destination type. Other conversions between arithmetic types may not always be able to represent the same value exactly:

* If a negative integer value is converted to an unsigned type, the resulting value corresponds to its 2's complement bitwise representation (i.e., -1 becomes the largest value representable by the type, -2 the second largest, ...).
* The conversions from/to bool consider false equivalent to *zero* (for numeric types) and to *null pointer* (for pointer types); true is equivalent to all other values and is converted to the equivalent of 1.
* If the conversion is from a floating-point type to an integer type, the value is truncated (the decimal part is removed). If the result lies outside the range of representable values by the type, the conversion causes *undefined behavior*.
* Otherwise, if the conversion is between numeric types of the same kind (integer-to-integer or floating-to-floating), the conversion is valid, but the value is *implementation-specific* (and may not be portable).

Some of these conversions may imply a loss of precision, which the compiler can signal with a warning. This warning can be avoided with an explicit conversion.  
  
For non-fundamental types, arrays and functions implicitly convert to pointers, and pointers in general allow the following conversions:

* *Null pointers* can be converted to pointers of any type
* Pointers to any type can be converted to void pointers.
* Pointer *upcast*: pointers to a derived class can be converted to a pointer of an *accessible* and *unambiguous* base class, without modifying its const or volatile qualification.

**Implicit conversions with classes**

In the world of classes, implicit conversions can be controlled by means of three member functions:

* **Single-argument constructors:** allow implicit conversion from a particular type to initialize an object.
* **Assignment operator:** allow implicit conversion from a particular type on assignments.
* **Type-cast operator:** allow implicit conversion to a particular type.

For example:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 | // implicit conversion of classes:  #include <iostream>  using namespace std;  class A {};  class B {  public:  // conversion from A (constructor):  B (const A& x) {}  // conversion from A (assignment):  B& operator= (const A& x) {return \*this;}  // conversion to A (type-cast operator)  operator A() {return A();}  };  int main ()  {  A foo;  B bar = foo; // calls constructor  bar = foo; // calls assignment  foo = bar; // calls type-cast operator  return 0;  } |

The type-cast operator uses a particular syntax: it uses the operator keyword followed by the destination type and an empty set of parentheses. Notice that the return type is the destination type and thus is not specified before theoperator keyword.

**Keyword explicit**

On a function call, C++ allows one implicit conversion to happen for each argument. This may be somewhat problematic for classes, because it is not always what is intended. For example, if we add the following function to the last example:

|  |  |  |
| --- | --- | --- |
|  | void fn (B arg) {} |  |

This function takes an argument of type B, but it could as well be called with an object of type A as argument:

|  |  |  |
| --- | --- | --- |
|  | fn (foo); |  |

This may or may not be what was intended. But, in any case, it can be prevented by marking the affected constructor with the explicit keyword:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 | // explicit:  #include <iostream>  using namespace std;  class A {};  class B {  public:  explicit B (const A& x) {}  B& operator= (const A& x) {return \*this;}  operator A() {return A();}  };  void fn (B x) {}  int main ()  {  A foo;  B bar (foo);  bar = foo;  foo = bar;    // fn (foo); // not allowed for explicit ctor.  fn (bar);  return 0;  } |

Additionally, constructors marked with explicit cannot be called with the assignment-like syntax; In the above example,bar could not have been constructed with:

|  |  |  |
| --- | --- | --- |
|  | B bar = foo; |  |

Type-cast member functions (those described in the previous section) can also be specified as explicit. This prevents implicit conversions in the same way as explicit-specified constructors do for the destination type.

**Type casting**

C++ is a strong-typed language. Many conversions, specially those that imply a different interpretation of the value, require an explicit conversion, known in C++ as *type-casting*. There exist two main syntaxes for generic type-casting:*functional* and *c-like*:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 | double x = 10.3;  int y;  y = int (x); // functional notation  y = (int) x; // c-like cast notation |  |

The functionality of these generic forms of type-casting is enough for most needs with fundamental data types. However, these operators can be applied indiscriminately on classes and pointers to classes, which can lead to code that -while being syntactically correct- can cause runtime [ERRORS](http://www.cplusplus.com/doc/tutorial/typecasting/#97915880). For example, the following code compiles without errors:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 | // class type-casting  #include <iostream>  using namespace std;  class Dummy {  double i,j;  };  class Addition {  int x,y;  public:  Addition (int a, int b) { x=a; y=b; }  int result() { return x+y;}  };  int main () {  Dummy d;  Addition \* padd;  padd = (Addition\*) &d;  cout << padd->result();  return 0;  } |

The program declares a pointer to Addition, but then it assigns to it a reference to an object of another unrelated type using explicit type-casting:

|  |  |  |
| --- | --- | --- |
|  | padd = (Addition\*) &d; |  |

Unrestricted explicit type-casting allows to convert any pointer into any other pointer type, independently of the types they point to. The subsequent call to member result will produce either a run-time error or some other unexpected results.  
  
In order to control these types of conversions between classes, we have four specific casting operators: dynamic\_cast,reinterpret\_cast, static\_cast and const\_cast. Their format is to follow the new type enclosed between angle-brackets (<>) and immediately after, the expression to be converted between parentheses.  
  
dynamic\_cast <new\_type> (expression)  
reinterpret\_cast <new\_type> (expression)  
static\_cast <new\_type> (expression)  
const\_cast <new\_type> (expression)  
  
The traditional type-casting equivalents to these expressions would be:  
  
(new\_type) expression  
new\_type (expression)  
  
but each one with its own special characteristics:

**dynamic\_cast**

dynamic\_cast can only be used with pointers and references to classes (or with void\*). Its purpose is to ensure that the result of the type conversion points to a valid complete object of the destination pointer type.  
  
This naturally includes *pointer upcast* (converting from pointer-to-derived to pointer-to-base), in the same way as allowed as an *implicit conversion*.  
  
But dynamic\_cast can also *downcast* (convert from pointer-to-base to pointer-to-derived) polymorphic classes (those with virtual members) if -and only if- the pointed object is a valid complete object of the target type. For example:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 | // dynamic\_cast  #include <iostream>  #include <exception>  using namespace std;  class Base { virtual void dummy() {} };  class Derived: public Base { int a; };  int main () {  try {  Base \* pba = new Derived;  Base \* pbb = new Base;  Derived \* pd;  pd = dynamic\_cast<Derived\*>(pba);  if (pd==0) cout << "Null pointer on first type-cast.\n";  pd = dynamic\_cast<Derived\*>(pbb);  if (pd==0) cout << "Null pointer on second type-cast.\n";  } catch (exception& e) {cout << "Exception: " << e.what();}  return 0;  } | Null pointer on second type-cast. |

|  |
| --- |
| **Compatibility note:** This type of dynamic\_cast requires *Run-Time Type Information (RTTI)* to keep track of dynamic types. Some compilers support this feature as an option which is disabled by default. This needs to be enabled for runtime type checking using dynamic\_cast to work properly with these types. |

The code above tries to perform two dynamic casts from pointer objects of type Base\* (pba and pbb) to a pointer object of type Derived\*, but only the first one is successful. Notice their respective initializations:

|  |  |  |
| --- | --- | --- |
| 1 2 | Base \* pba = new Derived;  Base \* pbb = new Base; |  |

Even though both are pointers of type Base\*, pba actually points to an object of type Derived, while pbb points to an object of type Base. Therefore, when their respective type-casts are performed using dynamic\_cast, pba is pointing to a full object of class Derived, whereas pbb is pointing to an object of class Base, which is an incomplete object of class Derived.  
  
When dynamic\_cast cannot cast a pointer because it is not a complete object of the required class -as in the second conversion in the previous example- it returns a *null pointer* to indicate the failure. If dynamic\_cast is used to convert to a reference type and the conversion is not possible, an exception of type bad\_cast is thrown instead.  
  
dynamic\_cast can also perform the other implicit casts allowed on pointers: casting null pointers between pointers types (even between unrelated classes), and casting any pointer of any type to a void\* pointer.

**static\_cast**

static\_cast can perform conversions between pointers to related classes, not only *upcasts* (from pointer-to-derived to pointer-to-base), but also *downcasts* (from pointer-to-base to pointer-to-derived). No checks are performed during runtime to guarantee that the object being converted is in fact a full object of the destination type. Therefore, it is up to the programmer to ensure that the conversion is safe. On the other side, it does not incur the overhead of the type-safety checks of dynamic\_cast.

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 | class Base {};  class Derived: public Base {};  Base \* a = new Base;  Derived \* b = static\_cast<Derived\*>(a); |  |

This would be valid code, although b would point to an incomplete object of the class and could lead to runtime ERRORS if de-referenced.  
  
Therefore, static\_cast is able to perform with pointers to classes not only the conversions allowed implicitly, but also their opposite conversions.  
  
static\_cast is also able to perform all conversions allowed implicitly (not only those with pointers to classes), and is also able to perform the opposite of these. It can:

* Convert from void\* to any pointer type. In this case, it guarantees that if the void\* value was obtained by converting from that same pointer type, the resulting pointer value is the same.
* Convert integers, floating-point values and enum types to enum types.

Additionally, static\_cast can also perform the following:

* Explicitly call a single-argument constructor or a conversion operator.
* Convert to *rvalue references*.
* Convert enum class values into integers or floating-point values.
* Convert any type to void, evaluating and discarding the value.

**reinterpret\_cast**

reinterpret\_cast converts any pointer type to any other pointer type, even of unrelated classes. The operation result is a simple binary copy of the value from one pointer to the other. All pointer conversions are allowed: neither the content pointed nor the pointer type itself is checked.  
  
It can also cast pointers to or from integer types. The format in which this integer value represents a pointer is platform-specific. The only guarantee is that a pointer cast to an integer type large enough to fully contain it (such as [intptr\_t](http://www.cplusplus.com/intptr_t)), is guaranteed to be able to be cast back to a valid pointer.  
  
The conversions that can be performed by reinterpret\_cast but not by static\_cast are low-level operations based on reinterpreting the binary representations of the types, which on most cases results in code which is system-specific, and thus non-portable. For example:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 | class A { /\* ... \*/ };  class B { /\* ... \*/ };  A \* a = new A;  B \* b = reinterpret\_cast<B\*>(a); |  |

This code compiles, although it does not make much sense, since now b points to an object of a totally unrelated and likely incompatible class. Dereferencing b is unsafe.

**const\_cast**

This type of casting manipulates the constness of the object pointed by a pointer, either to be set or to be removed. For example, in order to pass a const pointer to a function that expects a non-const argument:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 | // const\_cast  #include <iostream>  using namespace std;  void print (char \* str)  {  cout << str << '\n';  }  int main () {  const char \* c = "sample text";  print ( const\_cast<char \*> (c) );  return 0;  } | sample text |

The example above is guaranteed to work because function print does not write to the pointed object. Note though, that removing the constness of a pointed object to actually write to it causes *undefined behavior*.

**typeid**

typeid allows to check the type of an expression:   
  
typeid (expression)  
  
This operator returns a reference to a constant object of type [type\_info](http://www.cplusplus.com/type_info) that is defined in the standard header[<typeinfo>](http://www.cplusplus.com/%3Ctypeinfo%3E). A value returned by typeid can be compared with another value returned by typeid using operators == and!= or can serve to obtain a null-terminated character sequence representing the data type or class name by using itsname() member.

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 | // typeid  #include <iostream>  #include <typeinfo>  using namespace std;  int main () {  int \* a,b;  a=0; b=0;  if (typeid(a) != typeid(b))  {  cout << "a and b are of different types:\n";  cout << "a is: " << typeid(a).name() << '\n';  cout << "b is: " << typeid(b).name() << '\n';  }  return 0;  } | a and b are of different types:  a is: int \*  b is: int |

When typeid is applied to classes, typeid uses the RTTI to keep track of the type of dynamic objects. When typeid is applied to an expression whose type is a polymorphic class, the result is the type of the most derived complete object:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 | // typeid, polymorphic class  #include <iostream>  #include <typeinfo>  #include <exception>  using namespace std;  class Base { virtual void f(){} };  class Derived : public Base {};  int main () {  try {  Base\* a = new Base;  Base\* b = new Derived;  cout << "a is: " << typeid(a).name() << '\n';  cout << "b is: " << typeid(b).name() << '\n';  cout << "\*a is: " << typeid(\*a).name() << '\n';  cout << "\*b is: " << typeid(\*b).name() << '\n';  } catch (exception& e) { cout << "Exception: " << e.what() << '\n'; }  return 0;  } | a is: class Base \*  b is: class Base \*  \*a is: class Base  \*b is: class Derived |

*Note: The string returned by member name of [type\_info](http://www.cplusplus.com/type_info) depends on the specific implementation of your compiler and library. It is not necessarily a simple string with its typical type name, like in the compiler used to produce this output.*   
  
Notice how the type that typeid considers for pointers is the pointer type itself (both a and b are of type class Base \*). However, when typeid is applied to objects (like \*a and \*b) typeid yields their dynamic type (i.e. the type of their most derived complete object).  
  
If the type typeid evaluates is a pointer preceded by the dereference operator (\*), and this pointer has a null value,typeid throws a [bad\_typeid](http://www.cplusplus.com/bad_typeid) exception.

**Exceptions**

Exceptions provide a way to react to exceptional circumstances (like runtime[ERRORS[http://cdncache-a.akamaihd.net/items/it/img/arrow-10x10.png](http://www.cplusplus.com/doc/tutorial/exceptions/#41845946)](http://www.cplusplus.com/doc/tutorial/exceptions/#41845946)) in programs by transferring control to special functions called *handlers*.  
  
To catch exceptions, a portion of code is placed under exception inspection. This is done by enclosing that portion of code in a *try-block*. When an exceptional circumstance arises within that block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally and all handlers are ignored.  
  
An exception is thrown by using the throw keyword from inside the try block. Exception handlers are declared with the keyword catch, which must be placed immediately after the try block:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 | // exceptions  #include <iostream>  using namespace std;  int main () {  try  {  throw 20;  }  catch (int e)  {  cout << "An exception occurred. Exception Nr. " << e << '\n';  }  return 0;  } | An exception occurred. Exception Nr. 20 |

The code under exception handling is enclosed in a try block. In this example this code simply throws an exception:

|  |  |  |
| --- | --- | --- |
|  | throw 20; |  |

A throw expression accepts one parameter (in this case the integer value 20), which is passed as an argument to the exception handler.  
  
The exception handler is declared with the catch keyword immediately after the closing brace of the try block. The syntax for catch is similar to a regular function with one parameter. The type of this parameter is very important, since the type of the argument passed by the throw expression is checked against it, and only in the case they match, the exception is caught by that handler.  
  
Multiple handlers (i.e., catch expressions) can be chained; each one with a different parameter type. Only the handler whose argument type matches the type of the exception specified in the throw statement is executed.  
  
If an ellipsis (...) is used as the parameter of catch, that handler will catch any exception no matter what the type of the exception thrown. This can be used as a default handler that catches all exceptions not caught by other handlers:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 | try {  // code here  }  catch (int param) { cout << "int exception"; }  catch (char param) { cout << "char exception"; }  catch (...) { cout << "default exception"; } |  |

In this case, the last handler would catch any exception thrown of a type that is neither int nor char.  
  
After an exception has been handled the program, EXECUTION RESUMES after the *try-catch* block, not after the throw statement!.  
  
It is also possible to nest try-catch blocks within more external try blocks. In these cases, we have the possibility that an internal catch block forwards the exception to its external level. This is done with the expression throw; with no arguments. For example:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 | try {  try {  // code here  }  catch (int n) {  throw;  }  }  catch (...) {  cout << "Exception occurred";  } |  |

**Exception specification**

Older code may contain *dynamic exception specifications*. They are now deprecated in C++, but still supported. A *dynamic exception specification* follows the declaration of a function, appending a throw specifier to it. For example:

|  |  |  |
| --- | --- | --- |
|  | double myfunction (char param) throw (int); |  |

This declares a function called myfunction, which takes one argument of type char and returns a value of type double. If this function throws an exception of some type other than int, the function calls [std::unexpected](http://www.cplusplus.com/unexpected) instead of looking for a handler or calling [std::terminate](http://www.cplusplus.com/terminate).  
  
If this throw specifier is left empty with no type, this means that [std::unexpected](http://www.cplusplus.com/unexpected) is called for any exception. Functions with no throw specifier (regular functions) never call [std::unexpected](http://www.cplusplus.com/unexpected), but follow the normal path of looking for their exception handler.

|  |  |  |
| --- | --- | --- |
| 1 2 | int myfunction (int param) throw(); // all exceptions call unexpected  int myfunction (int param); // normal exception handling |  |

**Standard exceptions**

The C++ Standard library provides a base class specifically designed to declare objects to be thrown as exceptions. It is called [std::exception](http://www.cplusplus.com/exception) and is defined in the [<exception>](http://www.cplusplus.com/%3Cexception%3E) header. This class has a virtual member function called what that returns a null-terminated character sequence (of type char \*) and that can be overwritten in derived classes to contain some sort of description of the exception.

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 | // using standard exceptions  #include <iostream>  #include <exception>  using namespace std;  class myexception: public exception  {  virtual const char\* what() const throw()  {  return "My exception happened";  }  } myex;  int main () {  try  {  throw myex;  }  catch (exception& e)  {  cout << e.what() << '\n';  }  return 0;  } | My exception happened. |

We have placed a handler that catches exception objects by reference (notice the ampersand & after the type), therefore this catches also classes derived from exception, like our myex object of type myexception.  
  
All exceptions thrown by components of the C++ Standard library throw exceptions derived from this exception class. These are:

|  |  |
| --- | --- |
| **exception** | **description** |
| [bad\_alloc](http://www.cplusplus.com/bad_alloc) | thrown by new on allocation failure |
| [bad\_cast](http://www.cplusplus.com/bad_cast) | thrown by dynamic\_cast when it fails in a dynamic cast |
| [bad\_exception](http://www.cplusplus.com/bad_exception) | thrown by certain dynamic exception specifiers |
| [bad\_typeid](http://www.cplusplus.com/bad_typeid) | thrown by typeid |
| [bad\_function\_call](http://www.cplusplus.com/bad_function_call) | thrown by empty [function](http://www.cplusplus.com/function) objects |
| [bad\_weak\_ptr](http://www.cplusplus.com/bad_weak_ptr) | thrown by [shared\_ptr](http://www.cplusplus.com/shared_ptr) when passed a bad [weak\_ptr](http://www.cplusplus.com/weak_ptr) |

Also deriving from exception, header [<exception>](http://www.cplusplus.com/%3Cexception%3E) defines two generic exception types that can be inherited by custom exceptions to report ERRORS:

|  |  |
| --- | --- |
| **exception** | **description** |
| [logic\_error](http://www.cplusplus.com/logic_error) | ERROR related to the internal logic of the program |
| [runtime\_error](http://www.cplusplus.com/runtime_error) | error detected during runtime |

A typical example where standard exceptions need to be checked for is on memory allocation:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 | // bad\_alloc standard exception  #include <iostream>  #include <exception>  using namespace std;  int main () {  try  {  int\* myarray= new int[1000];  }  catch (exception& e)  {  cout << "Standard exception: " << e.what() << endl;  }  return 0;  } |

The exception that may be caught by the exception handler in this example is a bad\_alloc. Because bad\_alloc is derived from the standard base class exception, it can be caught (capturing by reference, captures all related classes).

**Preprocessor directives**

Preprocessor directives are lines included in the code of programs preceded by a hash sign (#). These lines are not program statements but directives for the *preprocessor*. The preprocessor examines the code before actual compilation of code begins and resolves all these directives before any code is actually generated by regular statements.  
  
These *preprocessor directives* extend only across a single line of code. As soon as a newline character is found, the preprocessor directive is ends. No semicolon (;) is expected at the end of a preprocessor directive. The only way a preprocessor directive can extend through more than one line is by preceding the newline character at the end of the line by a backslash (\).

**macro definitions (#define, #undef)**

To define preprocessor macros we can use #define. Its syntax is:  
  
#define identifier replacement  
  
When the preprocessor encounters this directive, it replaces any occurrence of identifier in the rest of the code byreplacement. This replacement can be an expression, a statement, a block or simply anything. The preprocessor does not understand C++ proper, it simply replaces any occurrence of identifier by replacement.

|  |  |  |
| --- | --- | --- |
| 1 2 3 | #define TABLE\_SIZE 100  int table1[TABLE\_SIZE];  int table2[TABLE\_SIZE]; |  |

After the preprocessor has replaced TABLE\_SIZE, the code becomes equivalent to:

|  |  |  |
| --- | --- | --- |
| 1 2 | int table1[100];  int table2[100]; |  |

#define can work also with parameters to define function macros:

|  |  |  |
| --- | --- | --- |
|  | #define getmax(a,b) a>b?a:b |  |

This would replace any occurrence of getmax followed by two arguments by the replacement expression, but also replacing each argument by its identifier, exactly as you would expect if it was a function:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 | // function macro  #include <iostream>  using namespace std;  #define getmax(a,b) ((a)>(b)?(a):(b))  int main()  {  int x=5, y;  y= getmax(x,2);  cout << y << endl;  cout << getmax(7,x) << endl;  return 0;  } | 5  7 |

Defined macros are not affected by block structure. A macro lasts until it is undefined with the #undef preprocessor directive:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 | #define TABLE\_SIZE 100  int table1[TABLE\_SIZE];  #undef TABLE\_SIZE  #define TABLE\_SIZE 200  int table2[TABLE\_SIZE]; |  |

This would generate the same code as:

|  |  |  |
| --- | --- | --- |
| 1 2 | int table1[100];  int table2[200]; |  |

Function macro definitions accept two special operators (# and ##) in the replacement sequence:  
The operator #, followed by a parameter name, is replaced by a string literal that contains the argument passed (as if enclosed between double quotes):

|  |  |  |
| --- | --- | --- |
| 1 2 | #define str(x) #x  cout << str(test); |  |

This would be translated into:

|  |  |  |
| --- | --- | --- |
|  | cout << "test"; |  |

The operator ## concatenates two arguments leaving no blank spaces between them:

|  |  |  |
| --- | --- | --- |
| 1 2 | #define glue(a,b) a ## b  glue(c,out) << "test"; |  |

This would also be translated into:

|  |  |  |
| --- | --- | --- |
|  | cout << "test"; |  |

Because preprocessor replacements happen before any C++ syntax check, macro definitions can be a tricky feature. But, be careful: code that relies heavily on complicated macros become less readable, since the syntax expected is on many occasions different from the normal expressions programmers expect in C++.

**Conditional inclusions (#ifdef, #ifndef, #if, #endif, #else and #elif)**

These directives allow to include or discard part of the code of a program if a certain condition is met.  
  
#ifdef allows a section of a program to be compiled only if the macro that is specified as the parameter has been defined, no matter which its value is. For example:

|  |  |  |
| --- | --- | --- |
| 1 2 3 | #ifdef TABLE\_SIZE  int table[TABLE\_SIZE];  #endif |  |

In this case, the line of code int table[TABLE\_SIZE]; is only compiled if TABLE\_SIZE was previously defined with #define, independently of its value. If it was not defined, that line will not be included in the program compilation.  
  
#ifndef serves for the exact opposite: the code between #ifndef and #endif directives is only compiled if the specified identifier has not been previously defined. For example:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 | #ifndef TABLE\_SIZE  #define TABLE\_SIZE 100  #endif  int table[TABLE\_SIZE]; |  |

In this case, if when arriving at this piece of code, the TABLE\_SIZE macro has not been defined yet, it would be defined to a value of 100. If it already existed it would keep its previous value since the #define directive would not be executed.  
  
The #if, #else and #elif (i.e., "else if") directives serve to specify some condition to be met in order for the portion of code they surround to be compiled. The condition that follows #if or #elif can only evaluate constant expressions, including macro expressions. For example:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 | #if TABLE\_SIZE>200  #undef TABLE\_SIZE  #define TABLE\_SIZE 200    #elif TABLE\_SIZE<50  #undef TABLE\_SIZE  #define TABLE\_SIZE 50    #else  #undef TABLE\_SIZE  #define TABLE\_SIZE 100  #endif    int table[TABLE\_SIZE]; |  |

Notice how the entire structure of #if, #elif and #else chained directives ends with #endif.

The behavior of #ifdef and #ifndef can also be achieved by using the special operators defined and !defined respectively in any #if or #elif directive:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 | #if defined ARRAY\_SIZE  #define TABLE\_SIZE ARRAY\_SIZE  #elif !defined BUFFER\_SIZE  #define TABLE\_SIZE 128  #else  #define TABLE\_SIZE BUFFER\_SIZE  #endif |  |

**Line control (#line)**

When we compile a program and some ERROR happens during the compiling process, the compiler shows an ERROR MESSAGE with references to the name of the file where the error happened and a line number, so it is easier to find the code generating the error.  
  
The #line directive allows us to control both things, the line numbers within the code files as well as the file name that we want that appears when an error takes place. Its format is:   
  
#line number "filename"  
  
Where number is the new line number that will be assigned to the next code line. The line numbers of successive lines will be increased one by one from this point on.  
  
"filename" is an optional parameter that allows to redefine the file name that will be shown. For example:

|  |  |  |
| --- | --- | --- |
| 1 2 | #line 20 "assigning variable"  int a?; |  |

This code will generate an error that will be shown as error in file "assigning variable", line 20.

**Error directive (#error)**

This directive aborts the compilation process when it is found, generating a compilation ERROR that can be specified as its parameter:

|  |  |  |
| --- | --- | --- |
| 1 2 3 | #ifndef \_\_cplusplus  #error A C++ compiler is required!  #endif |  |

This example aborts the compilation process if the macro name \_\_cplusplus is not defined (this macro name is defined by default in all C++ compilers).

**Source file inclusion (#include)**

This directive has been used assiduously in other sections of this tutorial. When the preprocessor finds an #includedirective it replaces it by the entire content of the specified header or file. There are two ways to use #include:

|  |  |  |
| --- | --- | --- |
| 1 2 | #include <header>  #include "file" |  |

In the first case, a *header* is specified between angle-brackets <>. This is used to include headers provided by the implementation, such as the headers that compose the standard library (iostream, string,...). Whether the headers are actually files or exist in some other form is *implementation-defined*, but in any case they shall be properly included with this directive.  
  
The syntax used in the second #include uses quotes, and includes a *file*. The *file* is searched for in an *implementation-defined* manner, which generally includes the current path. In the case that the file is not found, the compiler interprets the directive as a *header* inclusion, just as if the quotes ("") were replaced by angle-brackets (<>).

**Pragma directive (#pragma)**

This directive is used to specify diverse options to the compiler. These options are specific for the platform and the compiler you use. Consult the manual or the reference of your compiler for more information on the possible parameters that you can define with #pragma.  
  
If the compiler does not support a specific argument for #pragma, it is ignored - no syntax error is generated.

**Predefined macro names**

The following macro names are always defined (they all begin and end with two underscore characters, \_):

|  |  |
| --- | --- |
| **macro** | **value** |
| \_\_LINE\_\_ | Integer value representing the current line in the source code file being compiled. |
| \_\_FILE\_\_ | A string literal containing the presumed name of the source file being compiled. |
| \_\_DATE\_\_ | A string literal in the form "Mmm dd yyyy" containing the date in which the compilation process began. |
| \_\_TIME\_\_ | A string literal in the form "hh:mm:ss" containing the time at which the compilation process began. |
| \_\_cplusplus | An integer value. All C++ compilers have this constant defined to some value. Its value depends on the version of the standard supported by the compiler:   * **199711L**: ISO C++ 1998/2003 * **201103L**: ISO C++ 2011   Non conforming compilers define this constant as some value at most five digits long. Note that many compilers are not fully conforming and thus will have this constant defined as neither of the values above. |
| \_\_STD\_HOSTED\_\_ | 1 if the implementation is a [*HOSTED*](http://www.cplusplus.com/doc/tutorial/preprocessor/#46704371) *implementation* (with all standard headers available) 0 otherwise. |

The following macros are optionally defined, generally depending on whether a feature is available:

|  |  |
| --- | --- |
| **macro** | **value** |
| \_\_STDC\_\_ | In C: if defined to 1, the implementation conforms to the C standard. In C++: Implementation defined. |
| \_\_STDC\_VERSION\_\_ | In C:   * **199401L**: ISO C 1990, Ammendment 1 * **199901L**: ISO C 1999 * **201112L**: ISO C 2011   In C++: Implementation defined. |
| \_\_STDC\_MB\_MIGHT\_NEQ\_WC\_\_ | 1 if multibyte encoding might give a character a different value in character literals |
| \_\_STDC\_ISO\_10646\_\_ | A value in the form yyyymmL, specifying the date of the Unicode standard followed by the encoding of wchar\_t characters |
| \_\_STDCPP\_STRICT\_POINTER\_SAFETY\_\_ | 1 if the implementation has *strict pointer safety* (see [get\_pointer\_safety](http://www.cplusplus.com/get_pointer_safety)) |
| \_\_STDCPP\_THREADS\_\_ | 1 if the program can have more than one thread |

Particular implementations may define additional constants.  
  
For example:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 | // standard macro names  #include <iostream>  using namespace std;  int main()  {  cout << "This is the line number " << \_\_LINE\_\_;  cout << " of file " << \_\_FILE\_\_ << ".\n";  cout << "Its compilation began " << \_\_DATE\_\_;  cout << " at " << \_\_TIME\_\_ << ".\n";  cout << "The compiler gives a \_\_cplusplus value of " << \_\_cplusplus;  return 0;  } | This is the line number 7 of file /home/jay/stdmacronames.cpp.  Its compilation began Nov 1 2005 at 10:12:29.  The compiler gives a \_\_cplusplus value of 1 |

# Input/output with files

C++ provides the following classes to perform output and input of characters to/from files:

* **ofstream:** Stream class to write on files
* **ifstream:** Stream class to read from files
* **fstream:** Stream class to both read and write from/to files.  
  These classes are derived directly or indirectly from the classes istream and ostream. We have already used objects whose types were these classes: cin is an object of class istream and cout is an object of class ostream. Therefore, we have already been using classes that are related to our file streams. And in fact, we can use our file streams the same way we are already used to use cin and cout, with the only difference that we have to associate these streams with physical files. Let's see an example:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 | // basic file operations  #include <iostream>  #include <fstream>  using namespace std;  int main () {  ofstream myfile;  myfile.open ("example.txt");  myfile << "Writing this to a file.\n";  myfile.close();  return 0;  } | [file example.txt]  Writing this to a file. |

This code creates a file called example.txt and inserts a sentence into it in the same way we are used to do with cout, but using the file stream myfile instead.  
  
But let's go step by step:

### Open a file

The first operation generally performed on an object of one of these classes is to associate it to a real file. This procedure is known as to *open a file*. An open file is represented within a program by a *stream* (i.e., an object of one of these classes; in the previous example, this was myfile) and any input or output operation performed on this stream object will be applied to the physical FILE ASSOCIATED  to it.  
  
In order to open a file with a stream object we use its member function open:  
  
open (filename, mode);  
  
Where filename is a string representing the name of the file to be opened, and mode is an optional parameter with a combination of the following flags:

|  |  |
| --- | --- |
| ios::in | Open for input operations. |
| ios::out | Open for output operations. |
| ios::binary | Open in binary mode. |
| ios::ate | Set the initial position at the end of the file. If this flag is not set, the initial position is the beginning of the file. |
| ios::app | All output operations are performed at the end of the file, appending the content to the current content of the file. |
| ios::trunc | If the file is opened for output operations and it already existed, its previous content is deleted and replaced by the new one. |

All these flags can be combined using the bitwise operator OR (|). For example, if we want to open the file example.bin in binary mode to add data we could do it by the following call to member function OPEN:

|  |  |  |
| --- | --- | --- |
| 1 2 | ofstream myfile;  myfile.open ("example.bin", ios::out | ios::app | ios::binary); |  |

Each of the open MEMBER FUNCTIONS OF CLASSES ofstream, ifstream and fstream has a default mode that is used if the file is opened without a second argument:

|  |  |
| --- | --- |
| **class** | **default mode parameter** |
| ofstream | ios::out |
| ifstream | ios::in |
| fstream | ios::in | ios::out |

For ifstream and ofstream classes, ios::in and ios::out are automatically and respectively assumed, even if a mode that does not include them is passed as second argument to the open member function (the flags are combined).  
  
For fstream, the default value is only applied if the function is called without specifying any value for the mode parameter. If the function is called with any value in that parameter the default mode is overridden, not combined.  
  
File streams opened in *binary mode* perform input and output operations independently of any format considerations. Non-binary files are known as *text files*, and some translations may occur due to formatting of some special characters (like newline and carriage return characters).  
  
Since the first task that is performed on a file stream is generally to open a file, these three classes include a constructor that automatically calls the open member function and has the exact same parameters as this member. Therefore, we could also have declared the previous myfile object and conduct the same opening operation in our previous example by writing:

|  |  |  |
| --- | --- | --- |
|  | ofstream myfile ("example.bin", ios::out | ios::app | ios::binary); |  |

Combining object construction and stream opening in a single statement. Both forms to open a file are valid and equivalent.  
  
To check if a file stream was successful opening a file, you can do it by calling to member is\_open. This member function returns a bool value of true in the case that indeed the stream object is associated with an open file, or false otherwise:

|  |  |  |
| --- | --- | --- |
|  | if (myfile.is\_open()) { /\* ok, proceed with output \*/ } |  |

### Closing a file

When we are finished with our input and output operations on a file we shall close it so that the OPERATING SYSTEM is notified and its resources become available again. For that, we call the stream's member function close. This member function takes flushes the associated buffers and closes the file:

|  |  |  |
| --- | --- | --- |
|  | myfile.close(); |  |

Once this member function is called, the stream object can be re-used to open another file, and the file is available again to be opened by other processes.  
  
In case that an object is destroyed while still associated with an open file, the destructor automatically calls the member function close.

### Text files

Text file streams are those where the ios::binary flag is not included in their opening mode. These files are designed to store text and thus all values that are input or output from/to them can suffer some formatting transformations, which do not necessarily correspond to their literal binary value.  
  
Writing operations on text files are performed in the same way we operated with cout:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 | // writing on a text file  #include <iostream>  #include <fstream>  using namespace std;  int main () {  ofstream myfile ("example.txt");  if (myfile.is\_open())  {  myfile << "This is a line.\n";  myfile << "This is another line.\n";  myfile.close();  }  else cout << "Unable to open file";  return 0;  } | [file example.txt]  This is a line.  This is another line. |

Reading from a file can also be performed in the same way that we did with cin:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 | // reading a text file  #include <iostream>  #include <fstream>  #include <string>  using namespace std;  int main () {  string line;  ifstream myfile ("example.txt");  if (myfile.is\_open())  {  while ( getline (myfile,line) )  {  cout << line << '\n';  }  myfile.close();  }  else cout << "Unable to open file";  return 0;  } | This is a line.  This is another line. |

This last example reads a text file and prints out its content on the screen. We have created a while loop that reads the file line by line, using getline. The value returned by getline is a reference to the stream object itself, which when evaluated as a boolean expression (as in this while-loop) is true if the stream is ready for more operations, and false if either the end of the file has been reached or if some other ERROR occurred.

### Checking state flags

The following member functions exist to check for specific states of a stream (all of them return a bool value):

bad()

Returns true if a reading or writing operation fails. For example, in the case that we try to write to a file that is not open for writing or if the device where we try to write has no space left.

fail()

Returns true in the same cases as bad(), but also in the case that a format[ERROR[http://cdncache-a.akamaihd.net/items/it/img/arrow-10x10.png](http://www.cplusplus.com/doc/tutorial/files/#78111888)](http://www.cplusplus.com/doc/tutorial/files/#78111888) happens, like when an alphabetical character is extracted when we are trying to read an integer number.

eof()

Returns true if a file open for reading has reached the end.

good()

It is the most generic state flag: it returns false in the same cases in which calling any of the previous functions would return true. Note that good and bad are not exact opposites (good checks more state flags at once).

The member function clear() can be used to reset the state flags.

### get and put stream positioning

All i/o streams objects keep internally -at least- one internal position:  
  
ifstream, like istream, keeps an internal *get position* with the location of the element to be read in the next input operation.  
  
ofstream, like ostream, keeps an internal *put position* with the location where the next element has to be written.  
  
Finally, fstream, keeps both, the *get* and the *put position*, like iostream.  
  
These internal stream positions point to the locations within the stream where the next reading or writing operation is performed. These positions can be observed and modified using the following member functions:

#### tellg() and tellp()

THESE TWO MEMBER FUNCTIONS WITH NO PARAMETERS RETURN A VALUE OF THE MEMBER TYPE streampos, which is a type representing the current *get position* (in the case of tellg) or the *put position* (in the case oftellp).

#### seekg() and seekp()

These functions allow to change the location of the *get* and *put positions*. Both functions are overloaded with two different prototypes. The first form is:  
  
seekg ( position );  
seekp ( position );  
  
Using this prototype, the stream pointer is changed to the absolute position position (counting from the beginning of the file). The type for this parameter is streampos, which is the same type as returned by functions tellg and tellp.  
  
The other form for these functions is:  
  
seekg ( offset, direction );  
seekp ( offset, direction );  
  
Using this prototype, the *get* or *put position* is set to an offset value relative to some specific point determined by the parameter direction. offset is of type streamoff. And direction is of type seekdir, which is an *enumerated type* that determines the point from where offset is counted from, and that can take any of the following values:

|  |  |
| --- | --- |
| ios::beg | offset counted from the beginning of the stream |
| ios::cur | offset counted from the current position |
| ios::end | offset counted from the end of the stream |

The following example uses the member functions we have just seen to obtain the size of a file:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 | // obtaining file size  #include <iostream>  #include <fstream>  using namespace std;  int main () {  streampos begin,end;  ifstream myfile ("example.bin", ios::binary);  begin = myfile.tellg();  myfile.seekg (0, ios::end);  end = myfile.tellg();  myfile.close();  cout << "size is: " << (end-begin) << " bytes.\n";  return 0;  } | size is: 40 bytes. |

Notice the type we have used for variables begin and end:

|  |  |  |
| --- | --- | --- |
|  | streampos size; |  |

streampos is a specific type used for buffer and file positioning and is the type returned by file.tellg(). Values of this type can safely be subtracted from other values of the same type, and can also be converted to an integer type large enough to contain the size of the file.

These stream positioning functions use two particular types: streampos and streamoff. These types are also defined as member types of the stream class:

|  |  |  |
| --- | --- | --- |
| **Type** | **Member type** | **Description** |
| streampos | ios::pos\_type | Defined as fpos<mbstate\_t>. It can be converted to/from streamoff and can be added or subtracted values of these types. |
| streamoff | ios::off\_type | It is an alias of one of the fundamental integral types (such as int or long long). |

Each of the member types above is an alias of its non-member equivalent (they are the exact same type). It does not matter which one is used. The member types are more generic, because they are the same on all stream objects (even on streams using exotic types of characters), but the non-member types are widely used in existing code for historical reasons.

### Binary files

For binary files, reading and writing data with the extraction and insertion operators (<< and >>) and functions likegetline is not efficient, since we do not need to format any data and data is likely not formatted in lines.  
  
File streams include two member functions specifically designed to read and write binary data sequentially: write and read. The first one (write)IS A MEMBER FUNCTION OF ostream (inherited by ofstream). And read is a member function of istream (inherited by ifstream. OBJECTS OF CLASS fstream have both. Their prototypes are:  
  
write ( memory\_block, size );  
read ( memory\_block, size );  
  
Where memory\_block is of type char\* (pointer to char), and represents the address of an array of bytes where the read data elements are stored or from where the data elements to be written are taken. The size parameter is an integer value that specifies the number of characters to be read or written from/to the memory block.

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 | // reading an entire binary file  #include <iostream>  #include <fstream>  using namespace std;  int main () {  streampos size;  char \* memblock;  ifstream file ("example.bin", ios::in|ios::binary|ios::ate);  if (file.is\_open())  {  size = file.tellg();  memblock = new char [size];  file.seekg (0, ios::beg);  file.read (memblock, size);  file.close();  cout << "the entire file content is in memory";  delete[] memblock;  }  else cout << "Unable to open file";  return 0;  } | the entire file content is in memory |

In this example, the entire file is read and stored in a memory block. Let's examine how this is done:  
  
First, the file is open with the ios::ate flag, which means that the get pointer will be positioned at the end of the file. This way, when we call to member tellg(), we will directly obtain the size of the file.  
  
Once we have obtained the size of the file, we request the allocation of a memory block large enough to hold the entire file:

|  |  |  |
| --- | --- | --- |
|  | memblock = new char[size]; |  |

Right after that, we proceed to set the *get position* at the beginning of the file (remember that we opened the file with this pointer at the end), then we read the entire file, and finally close it:

|  |  |  |
| --- | --- | --- |
| 1 2 3 | file.seekg (0, ios::beg);  file.read (memblock, size);  file.close(); |  |

At this point we could operate with the data obtained from the file. But our program simply announces that the content of the file is in memory and then finishes.

### Buffers and Synchronization

When we operate with file streams, these are associated to an internal buffer object of type streambuf. This buffer object may represent a memory block that acts as an intermediary between the stream and the physical file. For example, with an ofstream, each time the member function put (which writes a single character) is called, the character may be inserted in this intermediate buffer instead of being written directly to the physical file with which the stream is associated.  
  
The OPERATING SYSTEM may also define other layers of buffering for reading and writing to files.  
  
When the buffer is flushed, all the data contained in it is written to the physical medium (if it is an output stream). This process is called *SYNCHRONIZATION* and takes place under any of the following circumstances:

* **When the file is closed:** before closing a file, all buffers that have not yet been flushed are synchronized and all pending data is written or read to the physical medium.
* **When the buffer is full:** Buffers have a certain size. When the buffer is full it is automatically synchronized.
* **Explicitly, with manipulators:** When certain manipulators are used on streams, an explicit synchronization takes place. These manipulators are: flush and endl.
* **Explicitly, with member function sync():** Calling the stream's member function sync() causes an immediate synchronization. This function returns an int value equal to -1 if the stream has no associated buffer or in case of failure. Otherwise (if the stream buffer was successfully synchronized) it returns 0.